Université Paris-Saclay
M2 Droit de la création et du numérique

Approche de l'élaboration et du fonctionnement des logiciels

Alain Delaët

- 6 séances de 3h en salle 109
  - cours magistral / parfois une partie de TP
  -  attention aux dates / horaires / salles
- cours les 24/01, 07/02, 14/02, 21/02, 07/03 et 14/03
- un **examen** sur table, date à venir

questions ⇒ `alain.delaet@inria.fr`

acquérir une **culture générale** concernant le logiciel
pas de polycopié, mais un **glossaire** et des **aide-mémoires**
(distribués en cours et par mail)

ce cours n'aborde **pas** les licences de logiciel

plan du cours

1 architecture des ordinateurs, programmes et données

2&3 faire l'expérience de la programmation

4 la complexité du logiciel

5 développer des logiciels

6 logiciels et Internet

7 bases de données

aujourd'hui

---

architecture des ordinateurs, programmes et données

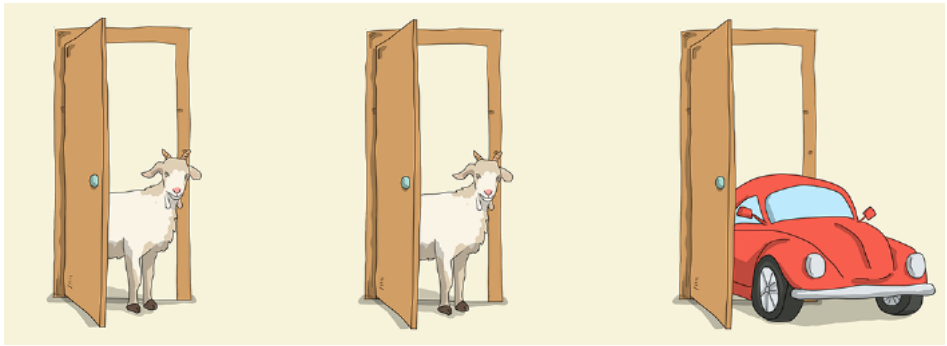
plan

1. un premier programme
2. octets et données
   - entiers
   - réels
   - textes
3. langage machine
4. langage de programmation
   - compilateur, interprète

le problème de Monty Hall

à l'origine, un jeu télévisé :

- devant vous, trois portes ; derrière une, il y une voiture, derrière les deux autres, une chèvre



(illustration San Tuon)

- vous choisissez une porte
- sans l'ouvrir, le présentateur ouvre une autre porte, derrière laquelle se trouve une chèvre (le présentateur connaît la bonne porte)
- vous avez maintenant la possibilité de choisir la troisième porte, ou de rester sur votre choix initial

toute la question est de savoir s'il y a stratégie meilleure qu'une autre

- faut-il changer de porte ?
- faut-il rester sur son choix ?
- peu importe ?

écrivons un programme pour simuler les deux stratégies changer/rester sur 1000 instances du jeu de Monty Hall
utilisons pour cela le langage **Python**, facile d'accès et idéal pour débuter la programmation (ce qu'on fera la semaine prochaine)

1. répéter 1000 fois
   - 1.1 tirer un nombre G au hasard entre 1 et 3 (la porte gagnante)
   - 1.2 tirer un nombre C au hasard entre 1 et 3 (la porte choisie)
   - 1.3 comparer ces deux nombres
     - si $C = G$, la stratégie « reste » gagne un point
     - si $C \neq G$, la stratégie « change » gagne un point
2. afficher les deux scores

```
from random import randint

n = 0 # nombre de simulations effectuées
change = 0 # score du joueur qui change de porte
reste  = 0 # score du joueur qui ne change pas

while n < 1000:
  g = randint(1, 3) # la porte gagnante
  c = randint(1, 3) # la porte choisie par le joueur
  if c == g:
    reste = reste + 1
  else:
    change = change + 1
    n = n+1

print(change, "victoires si on change")
print(reste, "victoires si on reste")
```

dans ce programme, on a

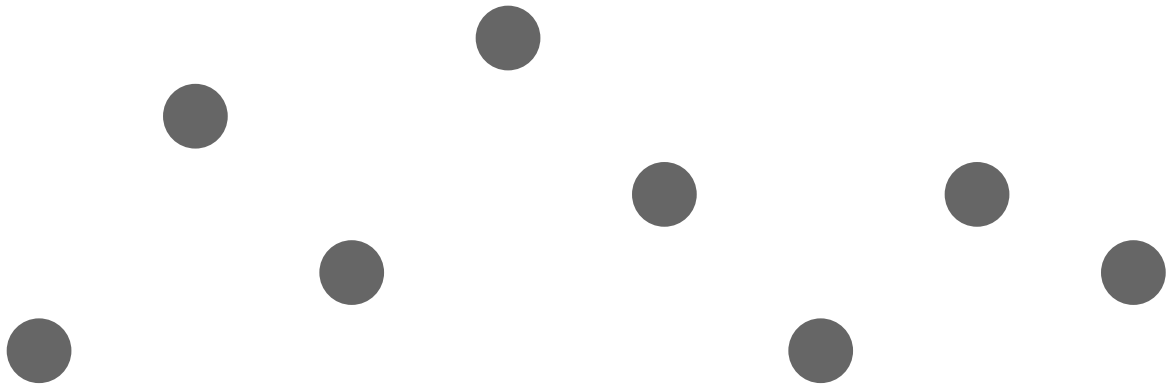
- utilisé des **variables** (`n`, `g`, `c`, etc.), c'est-à-dire des noms symboliques qui désignent des cases mémoire, contenant ici des entiers
- fait des **tests** (est-ce que $\texttt{c} = \texttt{g}$? est-ce que $\texttt{n} < 1000$?)
- répéter en **boucle** des opérations

c'est essentiellement

- des instructions
  - de calcul arithmétique élémentaire ($+$, $-$, $\times$, $\div$)
  - de contrôle (tester, aller ailleurs, revenir en arrière)
  - $\approx$ un milliard de fois par seconde
- de la mémoire
  - contient des données et des instructions
  - des milliards d'octets

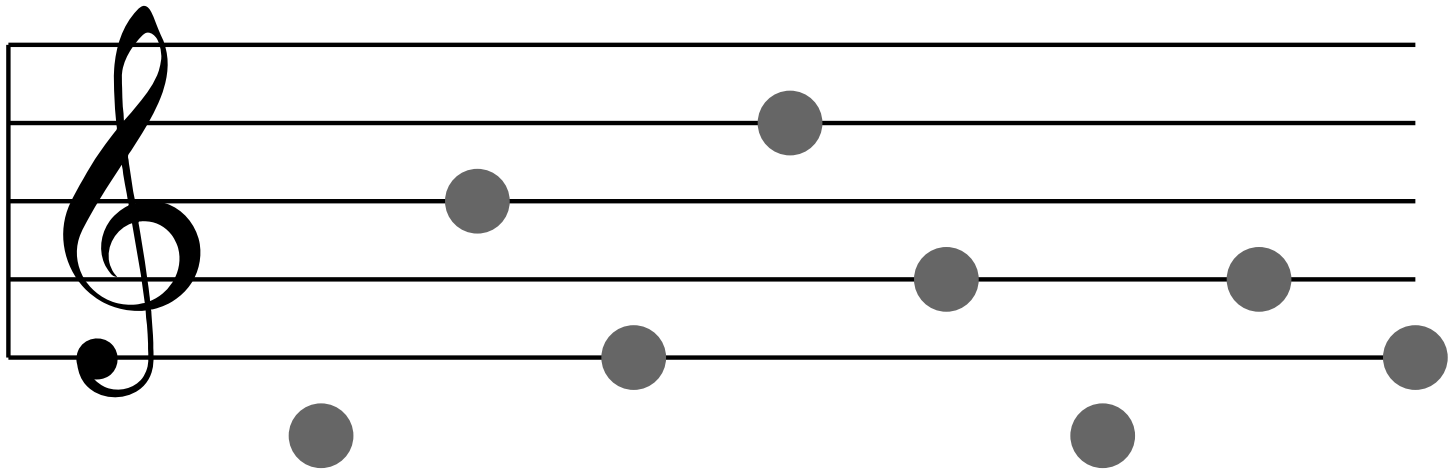
octets et données

Comment interpréter des données ?



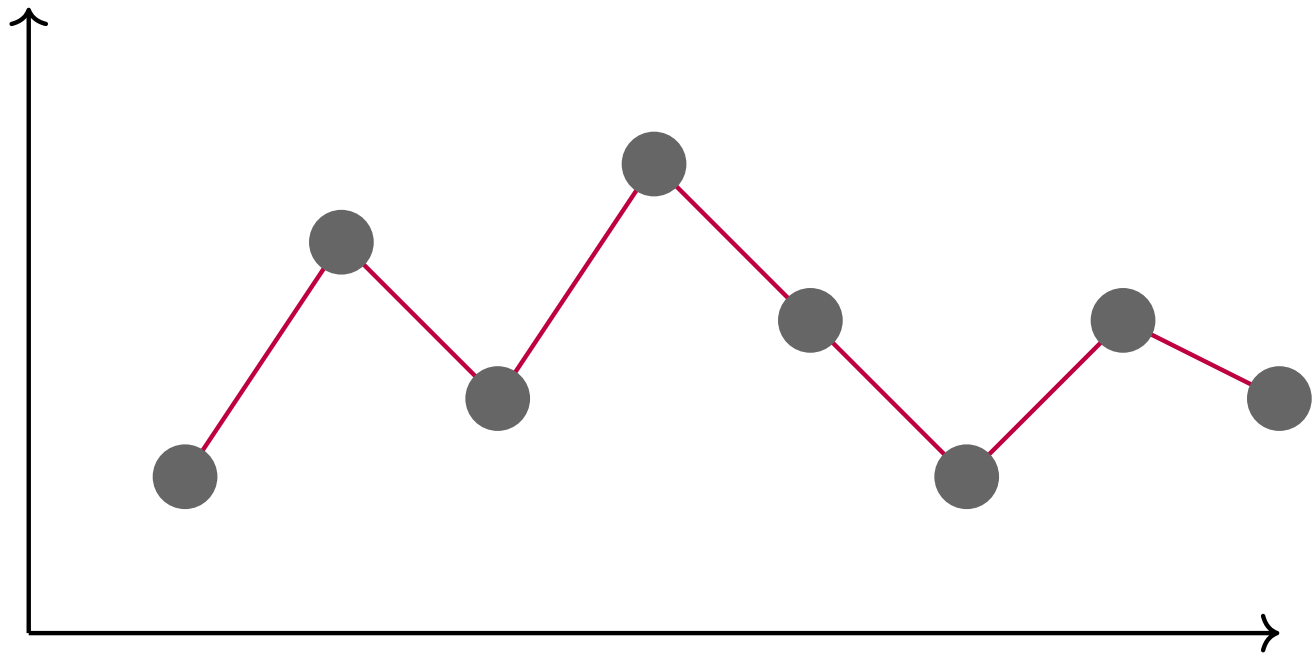
Wendy Mackay, leçon inaugurale au college de france

Une portée musicale ?



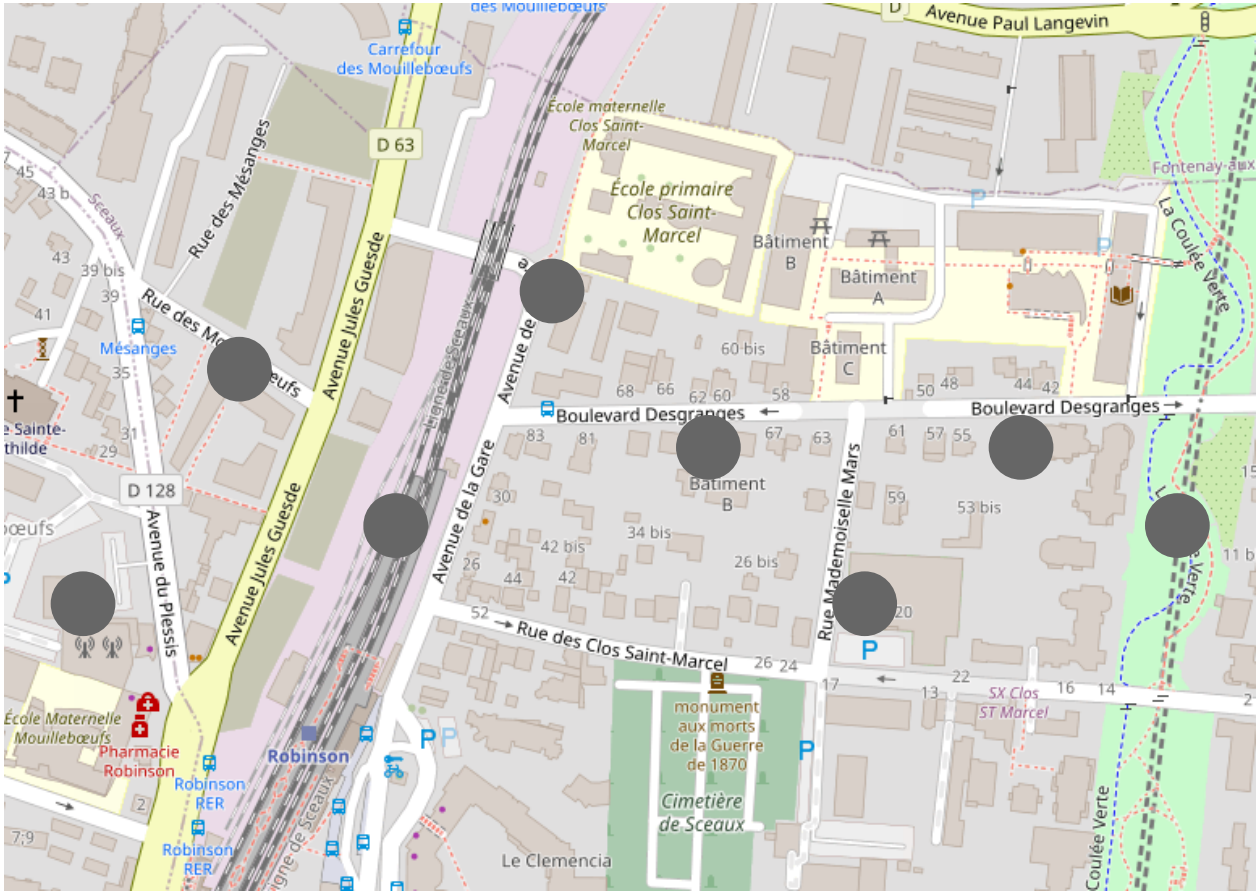
Wendy Mackay, leçon inaugurale au college de france

Un graphe ?



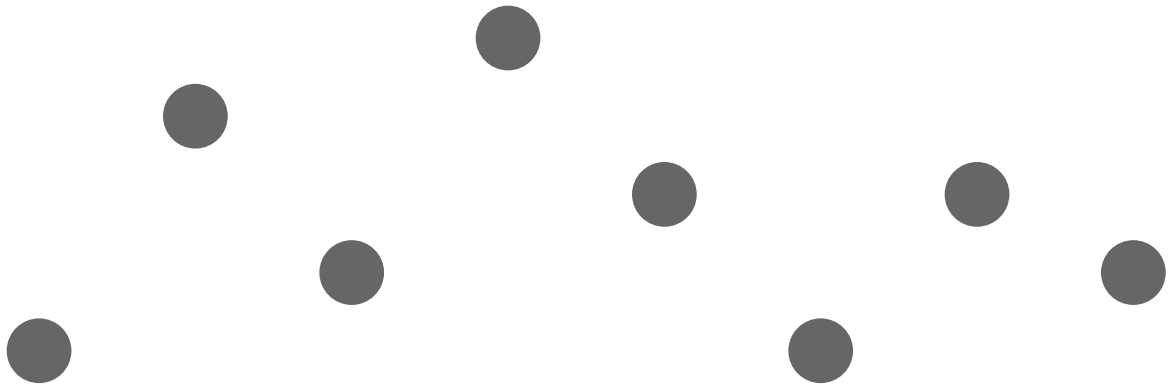
Wendy Mackay, leçon inaugurale au college de france

Une carte ?



Wendy Mackay, leçon inaugurale au college de france

Le substrat permet d'interpréter les données



Wendy Mackay, leçon inaugurale au college de france

avec des octets, on peut **représenter des données**
en particulier,

- des nombres entiers
- des nombres réels
- des textes

mais aussi,

- des images
- des vidéos

dans la machine, l'information est représentée sous la forme de 0 et de 1
(pas de courant / du courant)
ce sont en particulier les **chiffres de la base 2**
(en anglais, *bit* est la contraction de *binary digit* = chiffre binaire)

comme la base 10, mais avec uniquement deux chiffres

nombre en base 2	valeur en base 10
0	0
1	1
10	2
11	3
100	4
101	5
⋮	
101010	42
⋮	
$\underbrace{111\ldots111}_{64}$	18446744073709551615

quelle est l'écriture en base 2 de l'entier 85 ?

$$
\begin{aligned}
85 &= 64 + 16 + 4 + 1 \\
&= 2^6 + 2^4 + 2^2 + 2^0 \\
&= 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 1010101 \text{ en base } 2
\end{aligned}
$$

quel est l'entier qui s'écrit 1001001 en base 2 ?

$$
\begin{aligned}
1001001 &= 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 2^6 + 2^3 + 2^0 \\
&= 64 + 8 + 1 \\
&= 73
\end{aligned}
$$

	2^4	2^3	2^2	2^1	2^0	
		1	1	0	1	(13)
+			1	1	0	(6)
	1	0	0	1	1	(19)

dans la machine, un circuit réalise cette opération

	2^5	2^4	2^3	2^2	2^1	2^0	
				1	1	1	(7)
$\times$				1	1	0	(6)
				0	0	0	
			1	1	1		
		1	1	1			
$=$	1	0	1	0	1	0	(42)

dans la machine, un circuit réalise cette opération

dans la machine, les chiffres binaires ne sont pas manipulés individuellement, mais par paquets de 8 appelés **octets**
un octet peut prendre 256 valeurs différentes
($2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8$ combinaisons)

00000000, 00000001, 00000010, . . . , 11111111

soit les entiers de 0 à 255

la mémoire d'un ordinateur est un gigantesque ensemble de cases (des milliards !) contenant chacune un octet

0	0	42	200	17	0	0	73	255	...

(ici écrits en base 10)

chaque case porte un numéro, qu'on appelle une **adresse mémoire**

0	1	2	3	4	5	6	7	8	
0	0	42	200	17	0	0	73	255	...

ainsi, une instruction peut être

- mettre 42 dans la case 3
- ajouter 1 au contenu de la case 6
- etc.

on accède instantanément à n'importe quelle partie de la mémoire
et c'est pourquoi on parle de **RAM** (*Random Access Memory*)

pour l'instant, on ne parle que de la **mémoire vive**, c'est-à-dire la mémoire **volatile** dont le contenu est perdu lorsqu'elle n'est plus alimentée
on discutera plus loin de la mémoire non volatile

nom	symbole	valeur
kilooctet	ko	10^3 octets
megaoctet	Mo	10^6 octets
gigaoctet	Go	10^9 octets
teraoctet	To	10^{12} octets

note : historiquement, les multiples utilisés étaient des puissances de 2, car $2^{10} = 1024 \approx 1000$; on utilise Kio (kibioctet) pour désigner 2^{10} octets (et de même Mio, Gio et Tio pour 2^{20}, 2^{30} et 2^{40} octets)

représenter des nombres entiers

de manière immédiate, un octet représente un nombre entier entre 0 et 255
pour représenter un nombre plus grand, on peut utiliser plusieurs octets
exemple : le nombre 1 000 000 000 (un milliard) peut être représenté sur 4 octets, comme ceci

59	154	202	0

soit en binaire

00111011	10011010	11001010	00000000

les bases 2 ou 256 ne sont pas très pratiques
du coup, on utilise beaucoup la **base 16** dite **hexadécimale**

bits	hexa	valeur
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

bits	hexa	valeur
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

l'entier 1 000 000 000 s'écrit 3B9ACA00 en base 16

$$\underbrace{0011}_{3}\,\underbrace{1011}_{B}\,\underbrace{1001}_{9}\,\underbrace{1010}_{A}\,\underbrace{1100}_{C}\,\underbrace{1010}_{A}\,\underbrace{0000}_{0}\,\underbrace{0000}_{0}$$

note : un octet s'écrit donc sur deux chiffres hexadécimaux (de 00 à FF)


Nom de la Livebox :
Livebox - aaa1
Clé de sécurité :
E7A9 67F1 F596 1AEE 16EF 2F9F 9F
12V ⎓
Sagem
Livebox 2 FR
12345678
N° de série : 1234567890
@ Mac : AA:BB:CC:DD:EE:FF
1 234567 890123

ainsi, avec 4 octets (32 bits), on peut représenter tous les nombres entiers de 0 à $2^{32} - 1 \approx 4$ milliards

on dit que la machine est **32 bits** si elle sait faire des opérations arithmétiques sur des entiers de 32 bits

de même, avec 8 octets (64 bits), on peut représenter tous les nombres entiers de 0 à $2^{64} - 1 \approx 18$ milliards de milliards
on dit que la machine est **64 bits** si elle sait faire des opérations arithmétiques sur des entiers de 64 bits

que se passe-t-il si le résultat d'une opération est trop grand pour être représenté sur 32/64 bits ?
il est tout simplement **incorrect**, car des chiffres manquent
on parle de **débordement arithmétique**

sur une machine 32 bits, quand on calcule

$$8 \times 1\,000\,000\,000$$

on obtient

$$3\,705\,032\,704$$

le microprocesseur détecte et signale les débordements arithmétiques
(un drapeau est levé)
mais la quasi totalité des programmes ignorent ce signal, ce qui peut conduire à des catastrophes
[un exemple tristement célèbre : le vol inaugural d'Ariane 5, le 4 juin 1996]

représenter des nombres réels

on ne peut pas représenter tous les nombres réels dans un ordinateur
(il faudrait un temps de calcul et/ou une mémoire infinis pour représenter certains nombres)
conséquence : on représente les réels de façon **approximative**, c'est-à-dire avec une précision limitée

on s'inspire de la représentation scientifique pour approcher un réel sous la forme

$$\pm 1{,}f \times 2^e$$

où $1{,}f$ est la mantisse et e est l'exposant
exemples :

$$\begin{aligned} 0{,}5 &= 1 \times 2^{-1} \\ -214{,}75 &= -1{,}677734375 \times 2^7 \end{aligned}$$

on parle de **nombres à virgule flottante**, ou plus simplement de **flottants**

la **norme IEEE 754** définit le format des nombres à virgule flottante

	exposant (e)	fraction (f)	valeur
32 bits	8 bits	23 bits	$\pm 1,f \times 2^{e-127}$
64 bits	11 bits	52 bits	$\pm 1,f \times 2^{e-1023}$

on parle de flottants **simple précision** et **double précision** respectivement
les microprocesseurs supportent nativement les nombres flottants

la norme IEEE 754 spécifie également des modes d'**arrondi**
lorsque le résultat d'une opération n'est pas représentable, il est arrondi,
par exemple **au plus proche**

quand on écrit `1.6` dans un programme, le nombre flottant est en réalité

1,60000002384185791015625

car c'est là le flottant 32 bits le plus proche de 1,6

avec les flottants,

- les calculs sont souvent **inexacts**

$$\texttt{0.1 + 0.2} \neq \texttt{0.3}$$

- l'addition et la multiplication **ne sont pas associatives**

$$\texttt{1.6 + (3.2 + 1.7)} \neq \texttt{(1.6 + 3.2) + 1.7}$$

- la multiplication n'est **pas distributive par rapport à l'addition**

$$\texttt{1.5 * (3.2 + 1.4)} \neq \texttt{1.5*3.2 + 1.5*1.4}$$

en pratique, les impots avec de gros montants sont calculés à la main
d'autres représentations existent

- **GMP** Gnu Multiprecision Arithmetic
- permet de représenter les **entiers** et les **fractions**
- les opérations sont exactes, associatives, distributives

$$0.1 + 0.2 = 0.3$$

- sans erreur de débordement

$$8 \times 1\,000\,000\,000 = 8\,000\,000\,000$$

- mais les opérations sont plus lentes

module fractions en python.

représenter des textes

pour représenter des textes, on encode chaque caractère sur un ou plusieurs octets

de nombreux procédés ont été introduits, qui parfois coexistent

codage des caractères introduit dans les années 1960
(ASCII = *American Standard Code for Information Interchange*)
jeu de **128 caractères**, chacun étant codé sur un octet (0*bbbbbbb*)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

exemple : le caractère `A` a pour code 65 (soit 41 en hexa)

sous Windows, un retour chariot est représenté par deux caractères ASCII, à savoir CR (*Carriage Return*, code 13) et LF (*Line Feed*, code 10)
les systèmes Unix, dont Linux et macOS, utilisent uniquement LF (d'où parfois l'apparition de `^M` en fin de lignes en ouvrant un fichier Windows sous Linux)

l'ASCII est adapté à l'anglais, mais pas à la plupart des autres langues : il manque le é en français, le å en suédois, etc., sans même parler des langues non latines

dans les années 1980, on a introduit des jeux de 256 caractères pour différentes langues, compatibles avec l'ASCII

chaque caractère est toujours codé sur un octet

- les caractères 0 à 127 coïncident avec l'ASCII (0*bbbbbbb*)
- les caractères 128 à 255 sont spécifiques au jeu (1*bbbbbbb*)

code ISO	zone
8859-1 (latin-1)	Europe occidentale
8859-2 (latin-2)	Europe centrale ou de l'Est
8859-3 (latin-3)	Europe du Sud
8859-4 (latin-4)	Europe du Nord
8859-5	Cyrilique
⋮	

exemples : dans le jeu latin-1,

- le caractère `A` a pour code 65 (comme en ASCII)
- le caractère `é` a pour code 233

- on est limité à 256 caractères par jeu
  exemple : il manque le caractère œ dans latin-1
- on ne peux pas mélanger deux jeux de caractères
  exemple : « L'ukrainien (українська мова) est une langue parlée. »

la norme ISO-10646 ($\approx$ 1990) introduit un jeu de caractères universel (**UCS**, pour *Universal Character Set*) où chaque caractère porte

- un nom unique
- un numéro unique, appelé **point de code** (et noté `U+xxxx`)

exemples :

caractère	code	nom
A	`U+0041` (65)	Latin Capital letter A
é	`U+00E9` (233)	Latin Small Letter E with acute
Ӂ	`U+04C1` (1217)	Cyrillic Capital Letter Zhe with breve

note : les 256 premiers caractères coïncident avec latin-1

Unicode est une organisation privée à but non lucratif, qui introduit un standard pour encoder le jeu universel vers des octets
définit des **formats de transformation universelle** (**UTF**, pour *Universal Transformation Format*) : UTF-8, UTF-16 et UTF-32

chaque caractère est codé sur 8 bits minimum (c'est le sens du 8)

plage UCS	suite d'octets (en binaire)	bits
U+0000 à U+007F	**0**bbbbbbb	7 bits
U+0080 à U+07FF	**110**bbbbb **10**bbbbbb	11 bits
U+0800 à U+FFFF	**1110**bbbb **10**bbbbbb **10**bbbbbb	16 bits
U+10000 à U+10FFFF	**11110**bbb **10**bbbbbb **10**bbbbbb **10**bbbbbb	21 bits

note : compatible avec ASCII mais pas avec latin-1 (les 256 premiers points de code coïncident avec latin-1, mais pas leur encodage)

en UTF-8, le caractère é (Unicode 233 = 11101001) est codé sur deux octets

11000011 **10**101001

mais considérer (incorrectement) ces deux octets comme étant des caractères latin-1 donne

- le caractère 195 : Ã
- le caractère 169 : ©

les octets ne disent pas s'ils représentent des entiers, des réels ou encore des textes
ainsi, les quatre octets dans la machine

...	68	65	86	69	...

représentent aussi bien les caractères `DAVE`, que l'entier 1163280708 (sur 32 bits) ou encore le réel 3428,079102
c'est le **contexte** qui le détermine, selon comment on choisit de l'interpréter et ce qu'on choisit d'en faire

les octets représentent bien d'autres données encore
par exemple, une couleur peut être représentée sur trois octets dans le système **RGB** (pour *Red Green Blue*)

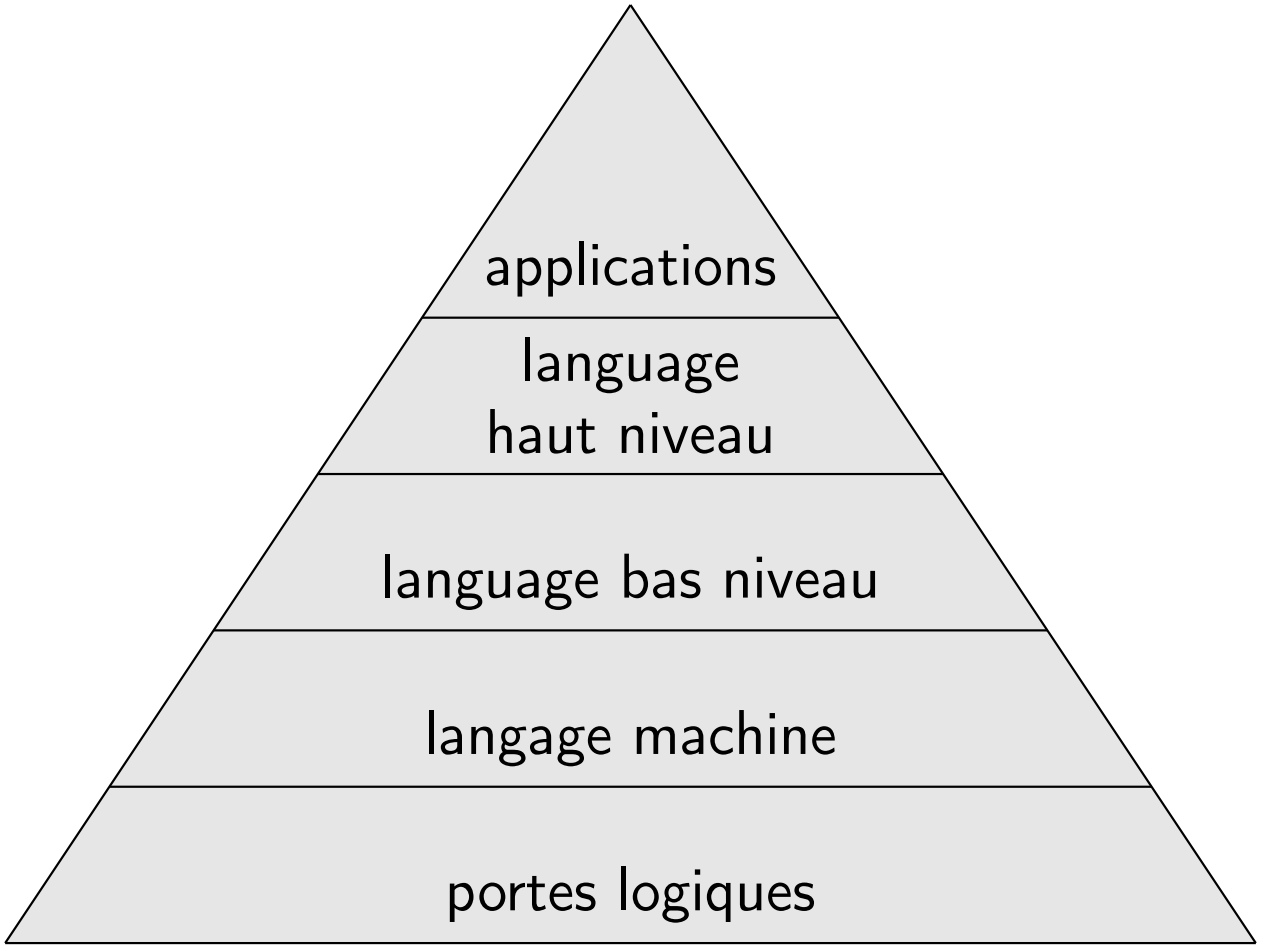
rouge	vert	bleu	not. hex.
11111111	11111111	11111111	#FFFFFF
11011011	01001100	01001100	#DB4C4C
10000000	10000000	10000000	#808080
01011101	10001010	10101000	#5D8AA8

soit $256 \times 256 \times 256 \approx 16$ millions de couleurs

les données n'ont pas de sens en dehors d'un contexte
tout est encodé avec des octets (nombres, texte, images, vidéos, documents, programmes)

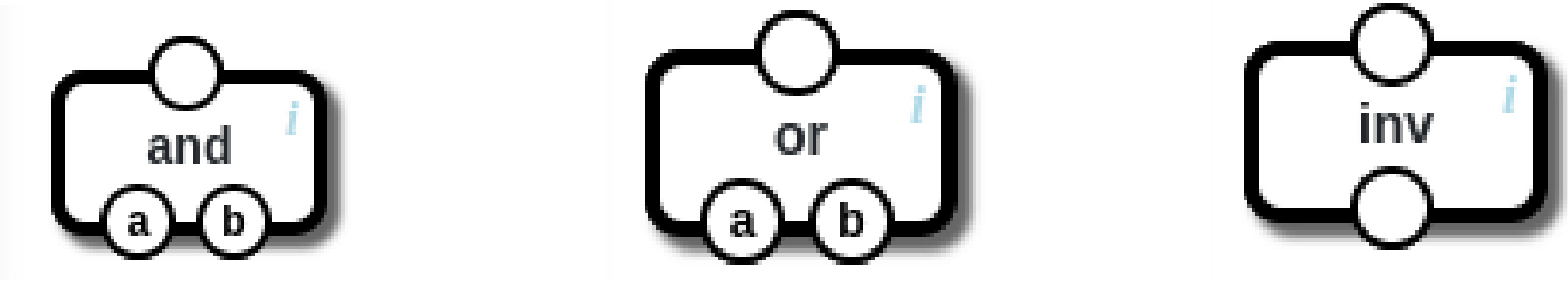
architecture des ordinateurs

hierarchie matérielle et logicielle



portes logiques

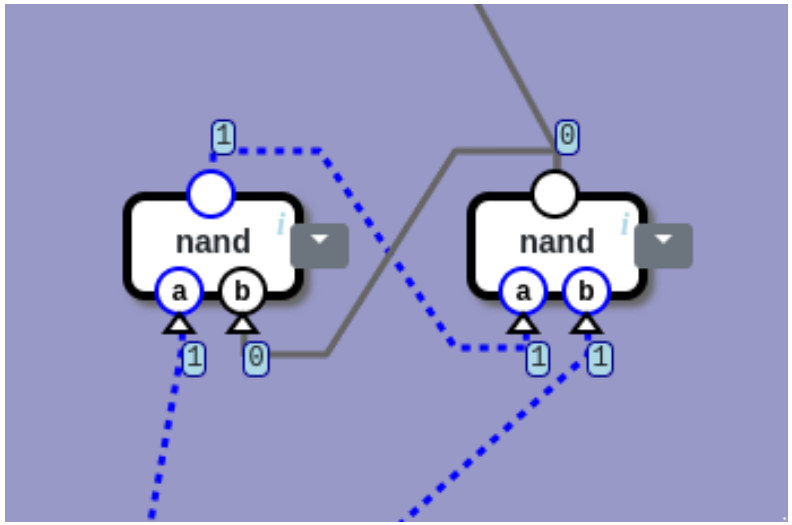
les portes logiques ont des entrées et des sorties



on presente leur comportement sous forme d'une **table de véritée**

a	b	a ou b	a et b	non a	non b
0	0	0	0	0	0
0	1	1	0	0	1
1	0	1	0	1	0
1	1	1	1	1	1

à partir de ces portes logiques, on peut implementer une memoire

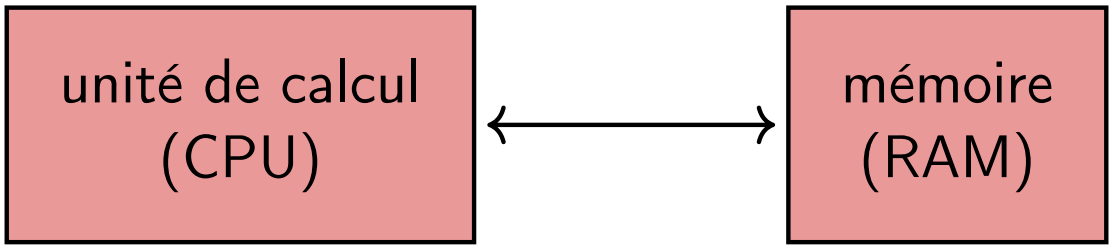


puis combiner les memoires entre elles pour avoir une plus grand memoire

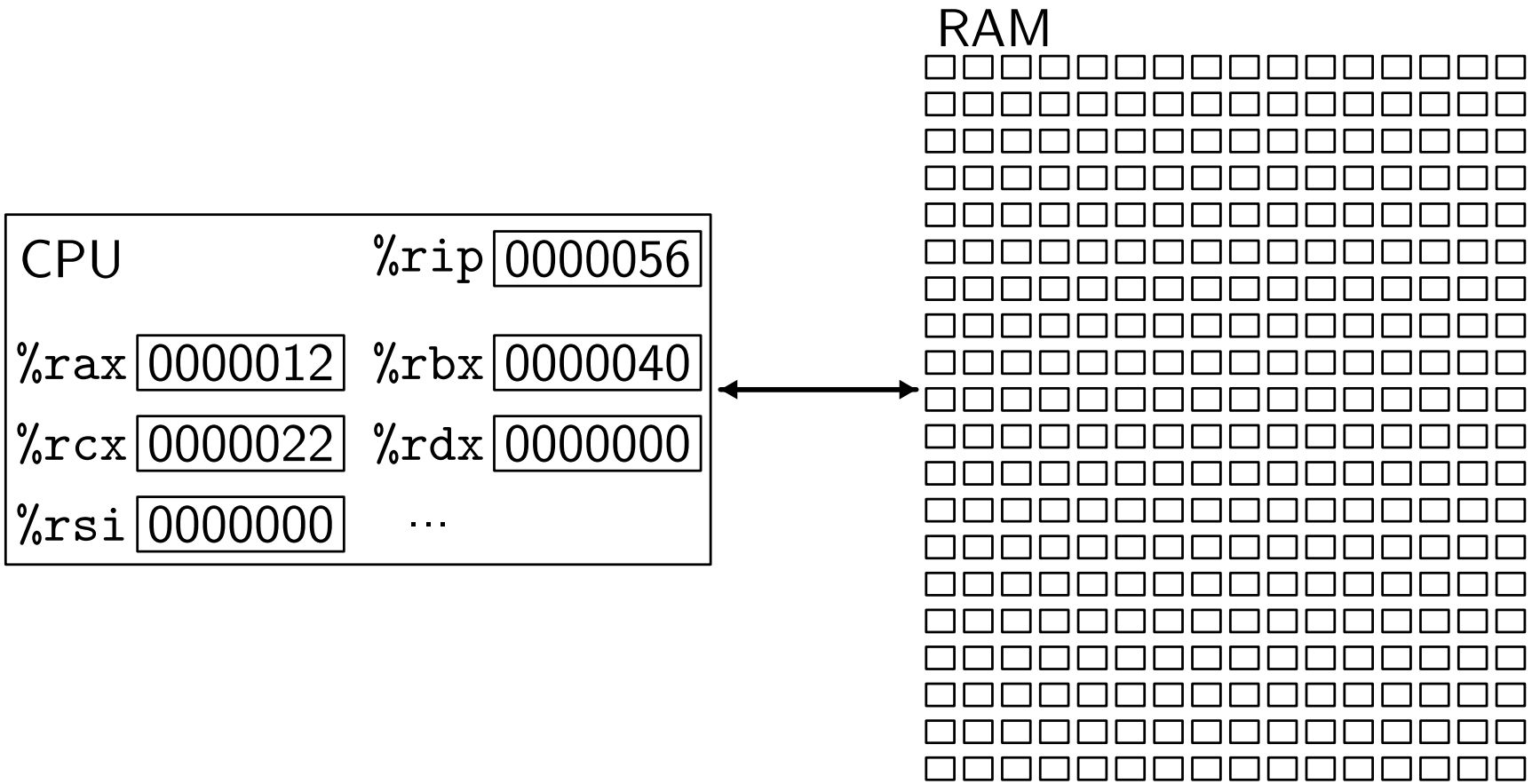
langage machine

les instructions aussi sont dans la mémoire (**modèle de von Neumann**)
elles sont codées par des séquences d'octets ; c'est le **langage machine**

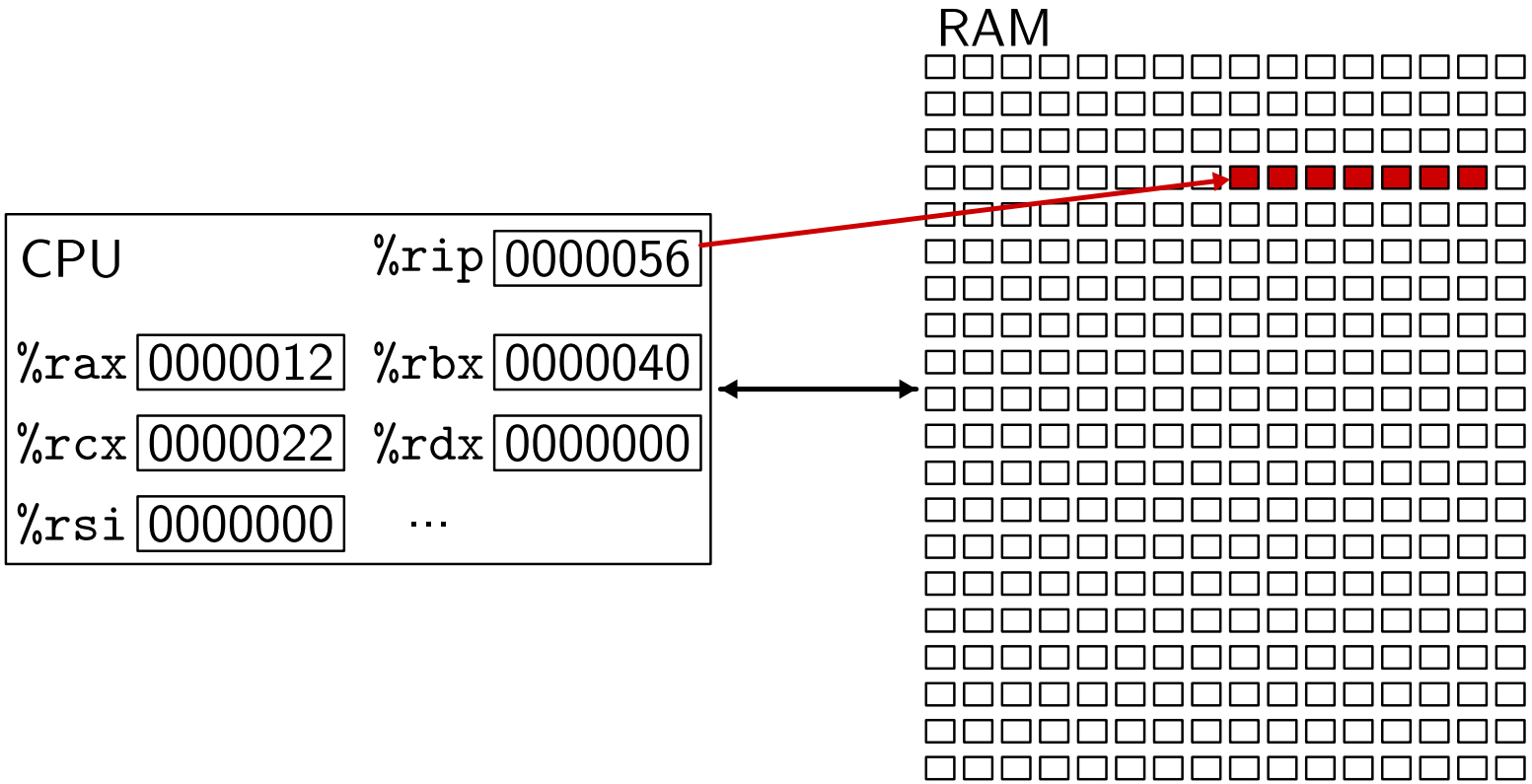
le CPU lit les instructions dans la mémoire, les décode et les exécute

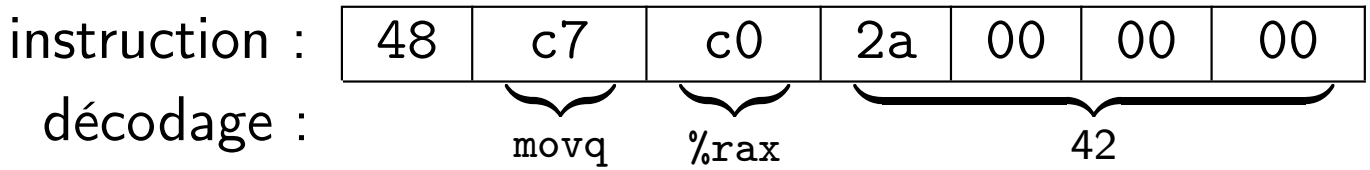


le CPU contient des **registres**, de petite taille et en petit nombre, qui servent au transfert de/vers la mémoire et aux calculs
sur une machine 64 bits, les registres ont une taille. . . 64 bits



le registre `%rip` contient l'adresse de la prochaine instruction à exécuter





i.e. mettre 42 dans le registre %rax

```
...
48 81 fb e8 03 00 00 // comparer le registre %rbx avec 1000
74 36                // si égalité, aller 54 octets plus loin
e8 ee fe ff ff       // appeler rand
48 99                // préparer une division
48 c7 c1 03 00 00 00 // mettre 3 dans le registre %rcx
48 f7 f9             // diviser %rdx:%rax par %rcx
...
```

ici, on a fait le choix arbitraire du registre %rbx pour stocker N
(on note en passant que la machine est petit-boutiste)

les octets ne disent pas s'ils représentent des instructions ou bien des données

ainsi, l'octet `C3` représente aussi bien l'instruction `ret` que l'entier 195

c'est le **contexte** qui le détermine, lorsqu'on choisit de lancer l'exécution à partir d'une certaine adresse

en pratique, on n'utilise pas le langage machine directement mais le **langage assembleur**

dans le langage assembleur, on a

- des noms symboliques pour les instructions : `mov`, `add`, etc.
- des étiquettes symboliques pour désigner des adresses mémoire
- des facilités pour écrire des entiers, des chaînes de caractères, etc.

le langage assembleur est **transformé** en langage machine,
par un programme (un compilateur), également appelé **assembleur**

```
...
cmpq  $1000, %rbx
  je    L
  call  rand
  cqto
  movq  $3, %rcx
idivq %rcx
...
L:...
```

cela reste un langage **très bas niveau**,
où une instruction assembleur = une instruction machine

langage de programmation

sauf cas exceptionnels, on ne programme plus les ordinateurs en assembleur,
mais dans des **langages de programmation** dits de **haut niveau**
ces langages fournissent des instructions/concepts qui n'existent pas dans la machine et qui vont être **traduits** en plusieurs instructions machine
exemples : C, C++, Java, Python, JavaScript, etc.
il y a aujourd'hui plus de 2500 langages de programmation !

le **langage C** est apparu en 1972, dans le contexte du développement du système d'exploitation Unix
encore massivement utilisé aujourd'hui, notamment pour le système

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n = 0; // nombre de simulations effectuées
    int change = 0, reste = 0;
    while (n < 1000) {
        int g = rand() % 3;
        int c = rand() % 3;
        if (c == g)
        reste++;
        else
        change++;
        n = n+1;
      }
    printf("le joueur qui change gagne %d fois\n", change);
    printf("le joueur qui reste  gagne %d fois\n", reste);
  }
```

comme en Python, on a

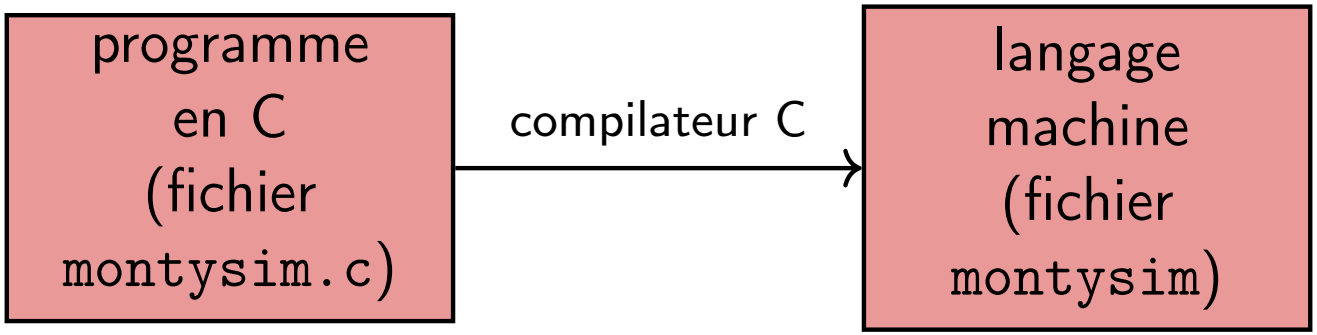
- chargé des **bibliothèques** (`stdio`,`stdlib`), qui fournissent ici les fonctions `printf` et `rand`
- utilisé des **variables** (`n`, `g`, `c`, etc.), c'est-à-dire des noms symboliques qui désignent des cases mémoire

contrairement à Python, on a

- déclaré des **types** (`int`) pour indiquer la nature des valeurs stockées dans nos variables

on note également des différences de syntaxe (parenthèses, accolades, etc.)

le programme C est traduit en langage machine par un **compilateur**

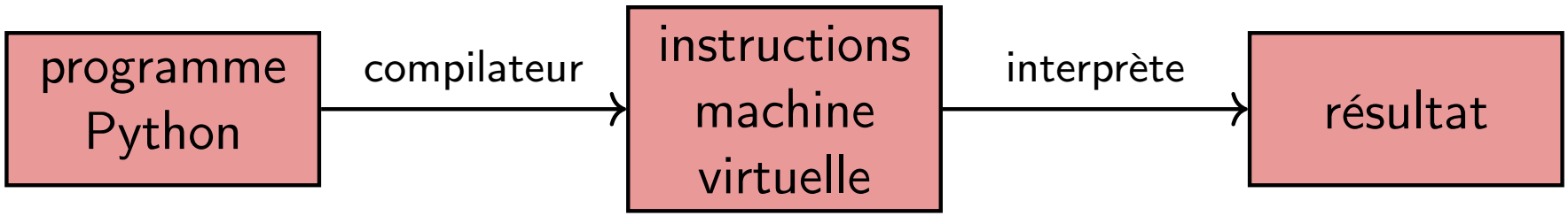


en particulier, le compilateur

- choisit des emplacements mémoire (registres, RAM) pour les variables
- traduit des constructions de haut niveau (boucle tant que)
- permet d'appeler des fonctions prédéfinies comme `printf`
- effectue un certain nombre de vérifications (on en reparlera)

pour le compilateur, le programme C est une donnée, manipulée en mémoire sous forme d'octets
et le résultat de l'exécution du compilateur est un programme (qui à son tour pourra être exécuté pour donner un résultat)

un programme Python n'est pas compilé vers le langage machine, mais vers une **machine virtuelle**, c'est-à-dire un ensemble d'instructions (inventées pour les seuls besoins de Python) qui sont **simulées** par un programme appelé **interprète**



on dit que Python est un **langage interprété**

Catala, un langage dédié à l'écriture d'implémentations correctes d'algorithmes dérivés de textes législatifs
voir `https://catala-lang.org/fr/`

activité

objectif : implementer une unité de calcul arithmétique (ALU)

- xor
- addition entre deux bits
- addition multi-bits
- incrément
- switch
- selector
- logic unit
- arithmetic unit
- arithmetic logic Unit

on le fait dans un navigateur :

`https://www.nandgame.com/`

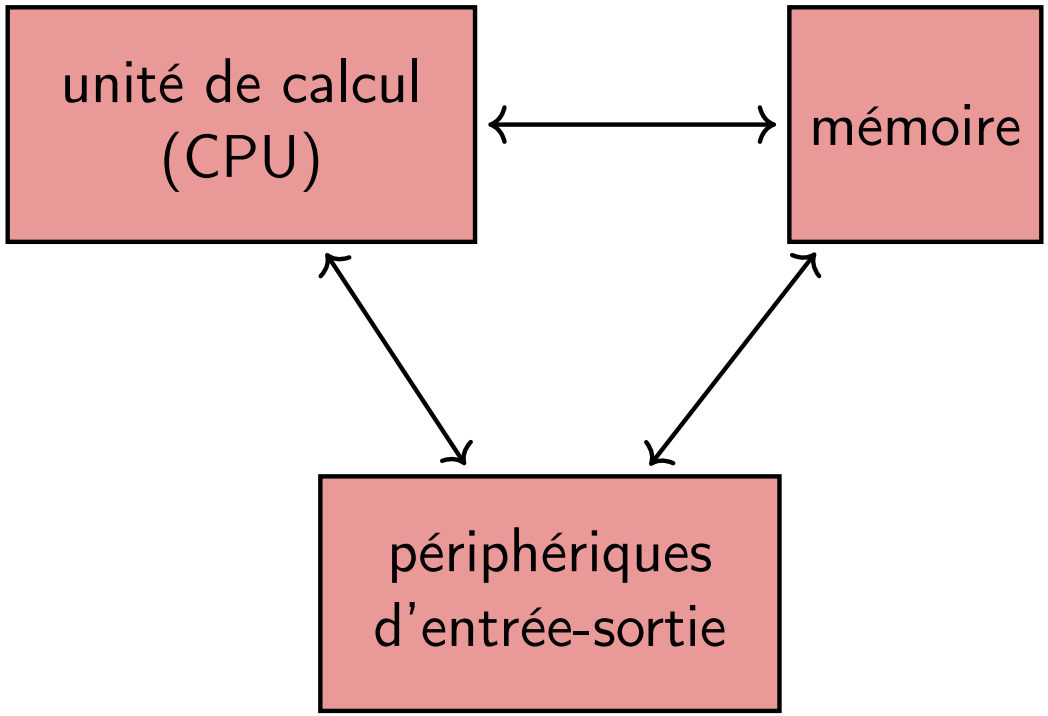
système d'exploitation

pour l'instant, on n'a pas parlé des périphériques

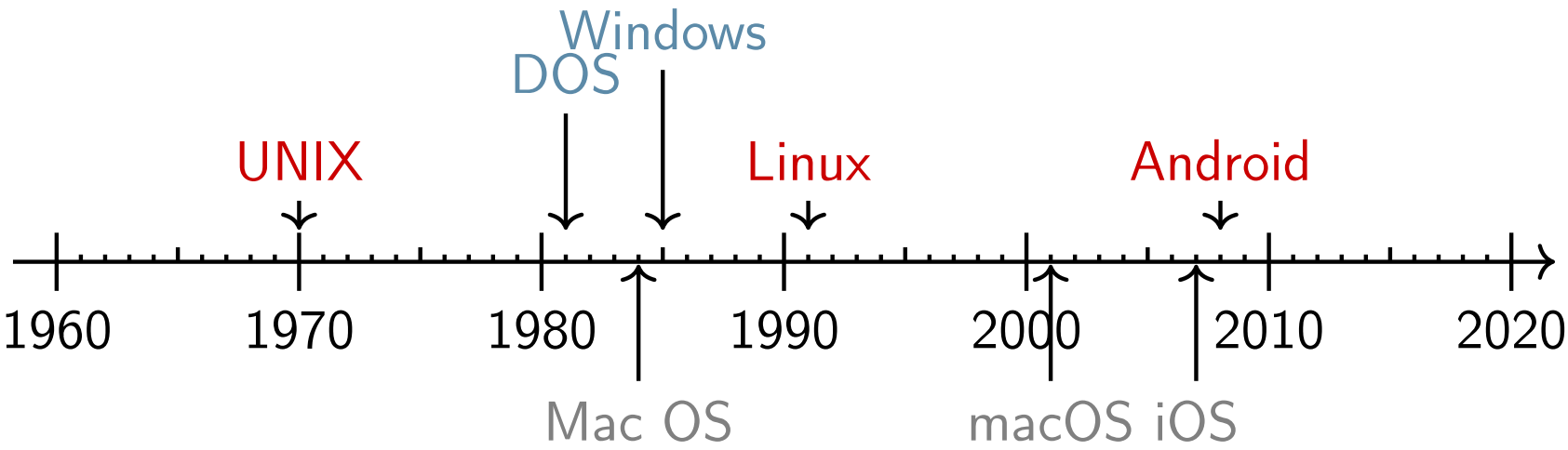
- clavier
- écran(s)
- disque dur
- carte réseau
- ports USB
- etc.

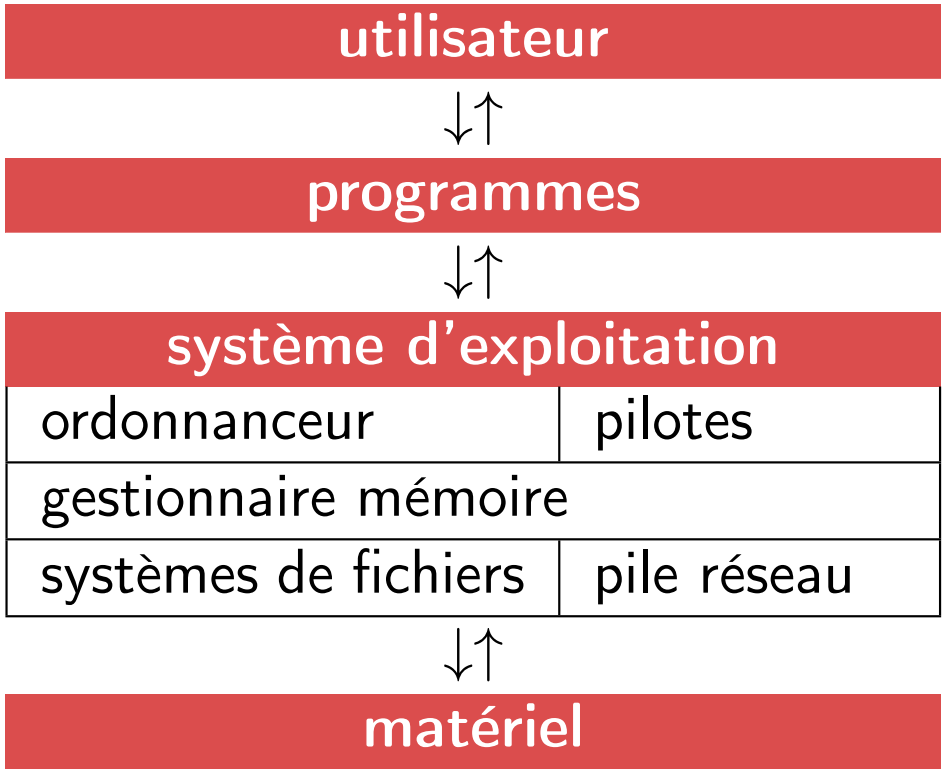
les périphériques

- lisent et écrivent des octets dans la mémoire
- se signalent auprès du CPU (interruption)

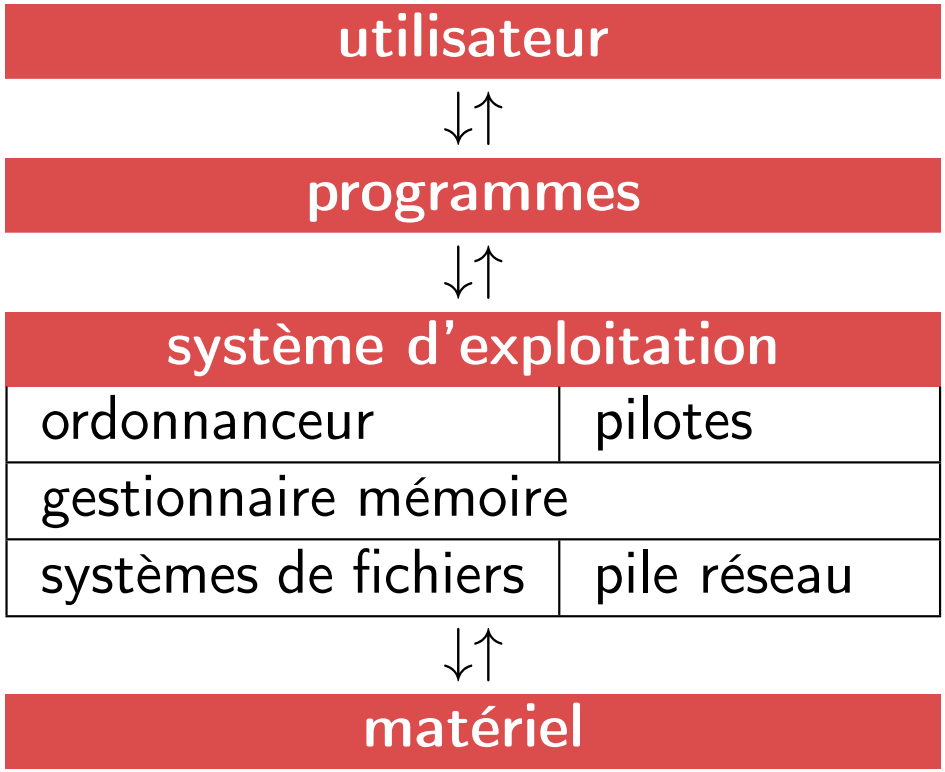


le programmeur n'interagit pas directement avec les périphériques, et plus généralement avec le matériel
il se repose pour cela sur le **système d'exploitation**, un ensemble de programmes qui gère les ressources matérielles et logicielles
exemples : Unix, Windows, Linux, macOS, Android, iOS, etc.

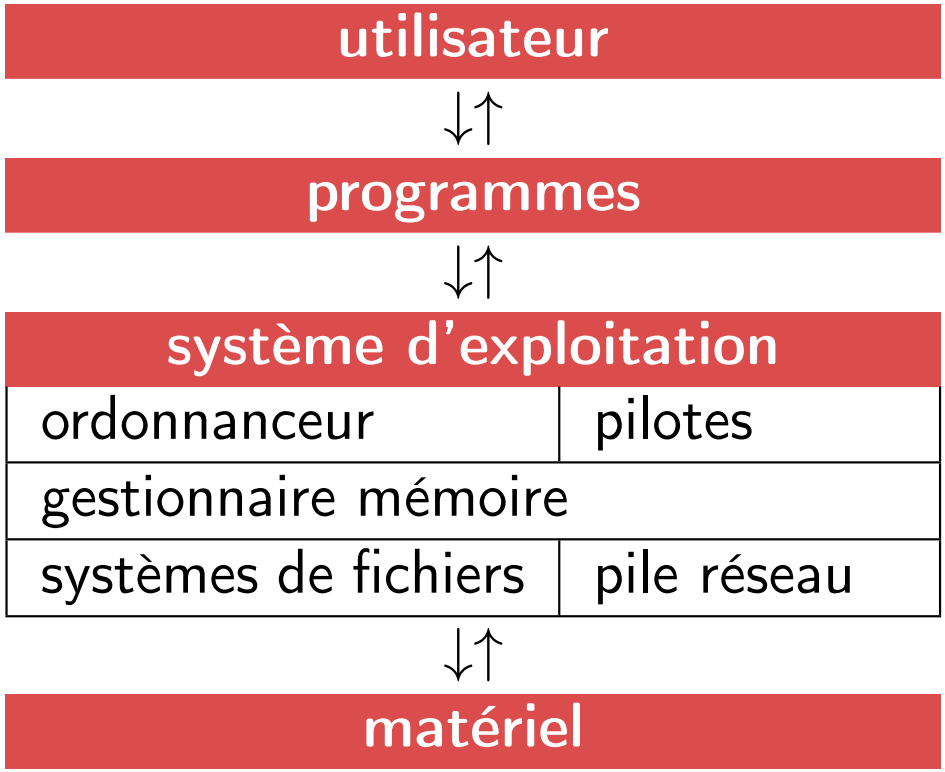

Windows
DOS
UNIX
Linux
Android
1960
1970
1980
1990
2000
2010
2020
Mac OS
macOS iOS



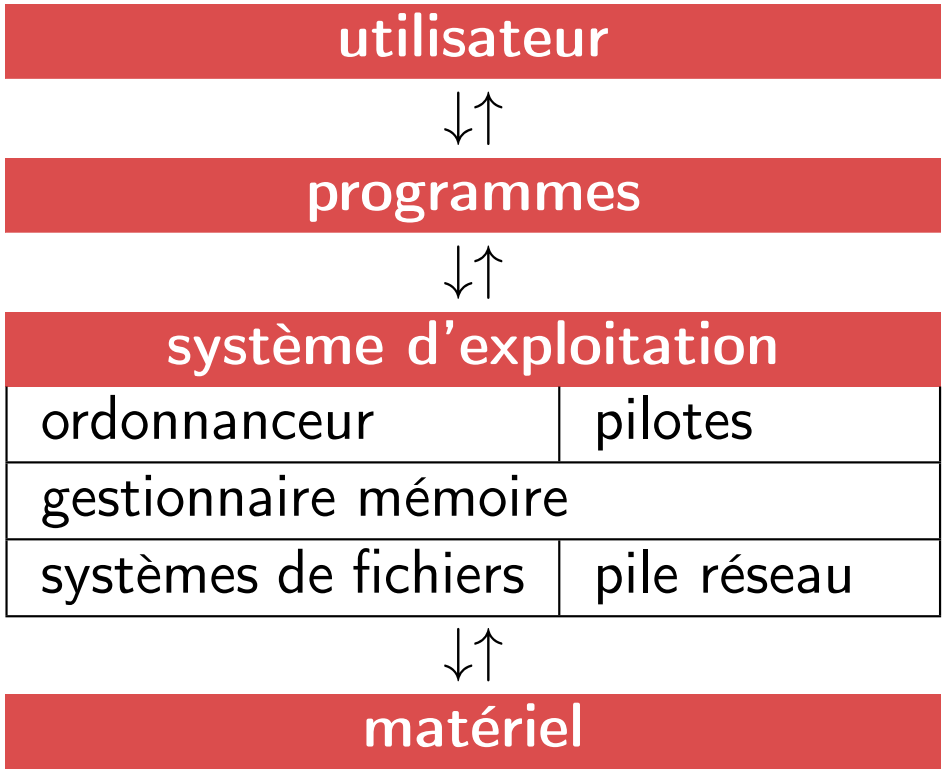
l'**ordonnanceur** décide quel programme s'exécute à un instant donné sur le processeur
donne l'illusion que plusieurs programmes s'exécutent simultanément sur la machine



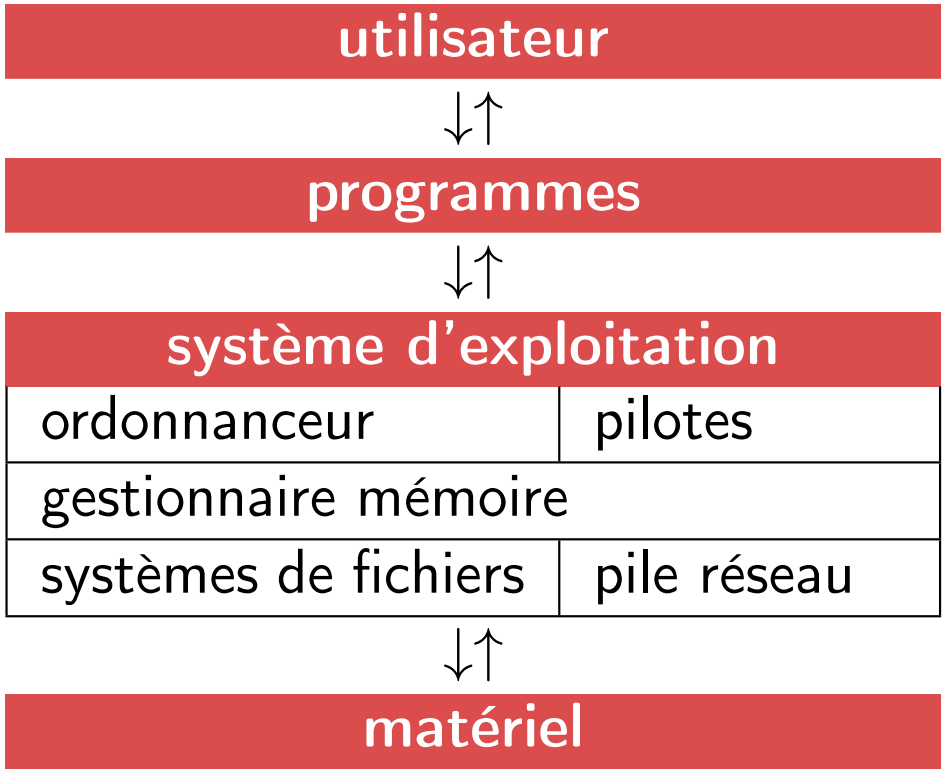
les **pilotes** de périphériques (en anglais *driver*) assurent l'interface avec les périphériques matériels : carte graphique, disque durs, clavier, etc.



le **gestionnaire de mémoire** répartit la mémoire vive entre les différents programmes en cours d'exécution



les différents **systèmes de fichiers** définissent la manière de stocker les fichiers sur les supports physiques : disques, clés USB, disques optiques, etc.
exemples : NTFS, ext4



la **pile réseau** implémente des protocoles réseau tels que TCP/IP (on en reparlera)

ensemble de standards définissant les fonctionnalités d'un système d'exploitation, et en particulier

- les fonctions de bibliothèque
- les programmes de base

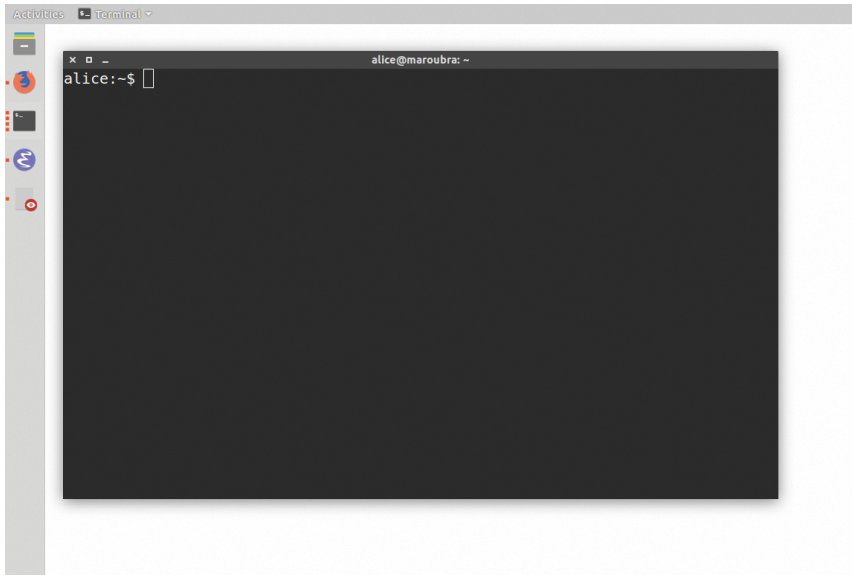
Linux, Android, macOS ou encore iOS sont tous compatibles POSIX
une exception notable : le système Windows

l'**interface système** (en anglais *shell*) est un programme permettant à l'utilisateur d'interagir avec le système d'exploitation

une boucle

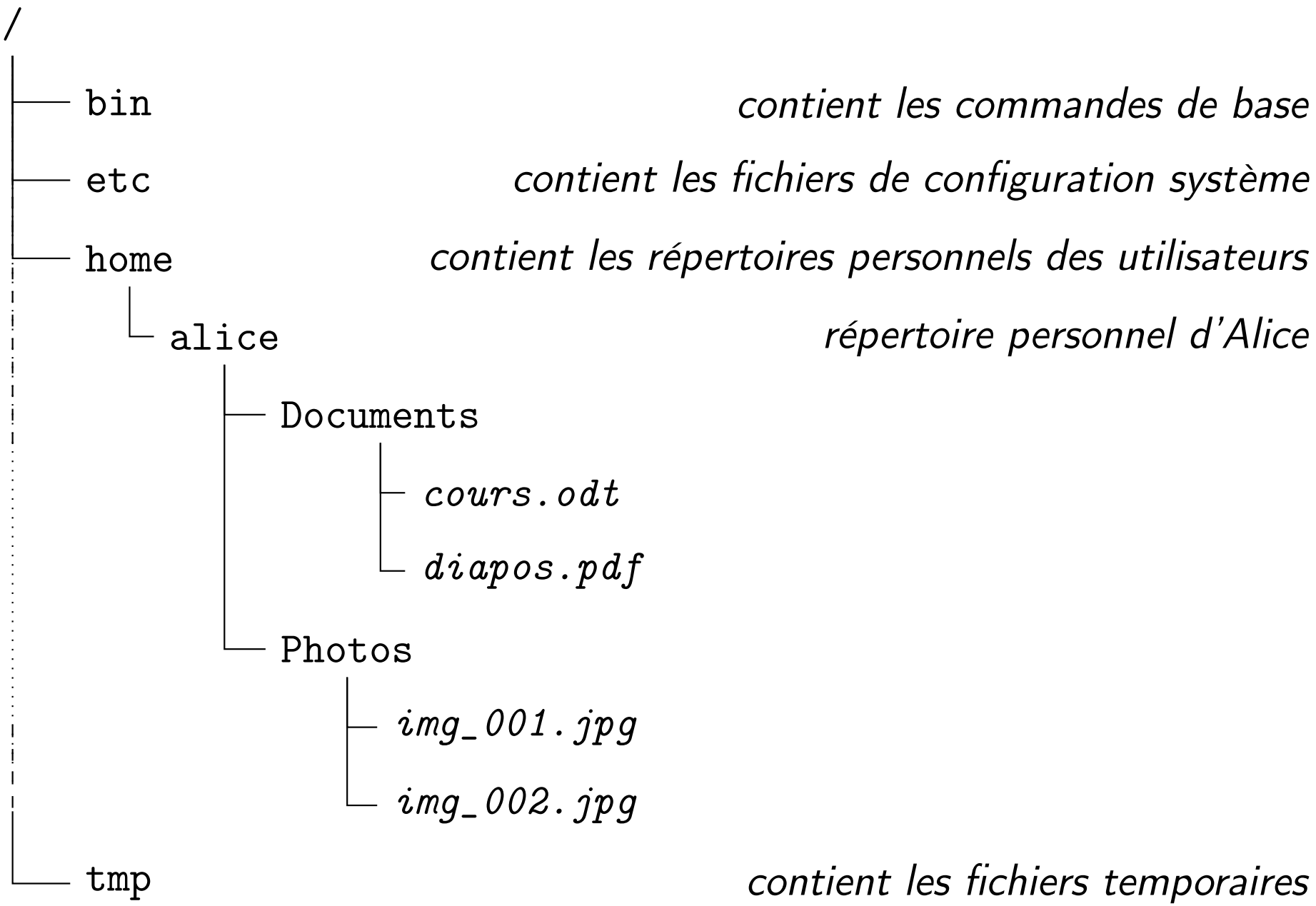
- saisir une commande
- l'exécuter

aujourd'hui dans un **émulateur de terminal**



les systèmes POSIX sont **multi-utilisateurs**
chaque utilisateur possède un **identifiant de connexion** (*login* en anglais), généralement associé à un mot de passe
correspond en interne à un identifiant numérique unique nommé UID (pour l'anglais *User IDentifier*)
un utilisateur spécial : `root`, le **super-utilisateur**, qui a tous les privilèges

l'ordinateur stocke sur sa mémoire **non volatile** (disque dur, clé USB, etc.)
des **fichiers** contenant des données ou des programmes
ces fichiers peuvent êtres lus, modifiés, renommés, copiés ou supprimés
ils sont **organisés** grâce à des répertoires (ou dossiers)


/
bin
contient les commandes de base
etc
contient les fichiers de configuration système
home
contient les répertoires personnels des utilisateurs
alice
répertoire personnel d'Alice
Documents
cours.odt
diapos.pdf
Photos
img_001.jpg
img_002.jpg
tmp
contient les fichiers temporaires

dans les systèmes POSIX, l'**extension** d'un nom de fichier n'a pas de signification particulière
l'utilisation de l'extension pour déterminer le type de fichier est une caractéristique du système Windows

- tout n'est qu'octets
- instructions et données dans la mémoire
- instructions finalement très primitives

-  **vendredi 7 février** 9h30 salle 109
- programmation Python 1/2